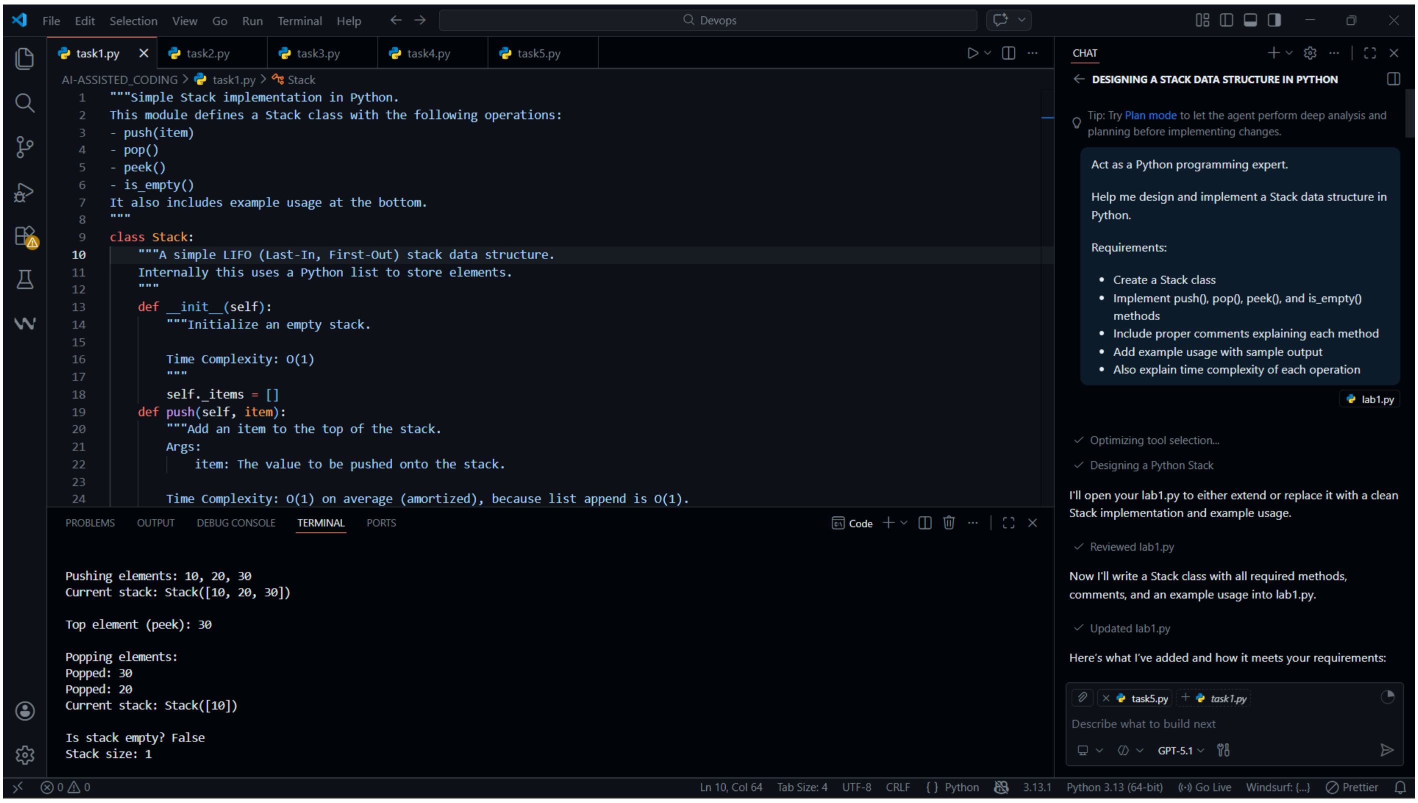
Assignment 11.2

Task Description -1 – (Stack Using AI Guidance)

• Task: With the help of AI, design and implement a Stack data structure supporting basic stack operations.

Expected Output:

• A Python Stack class supporting push, pop, peek, and empty-check operations with proper documentation.



Implemented a Stack class backed by a Python list.

Methods:

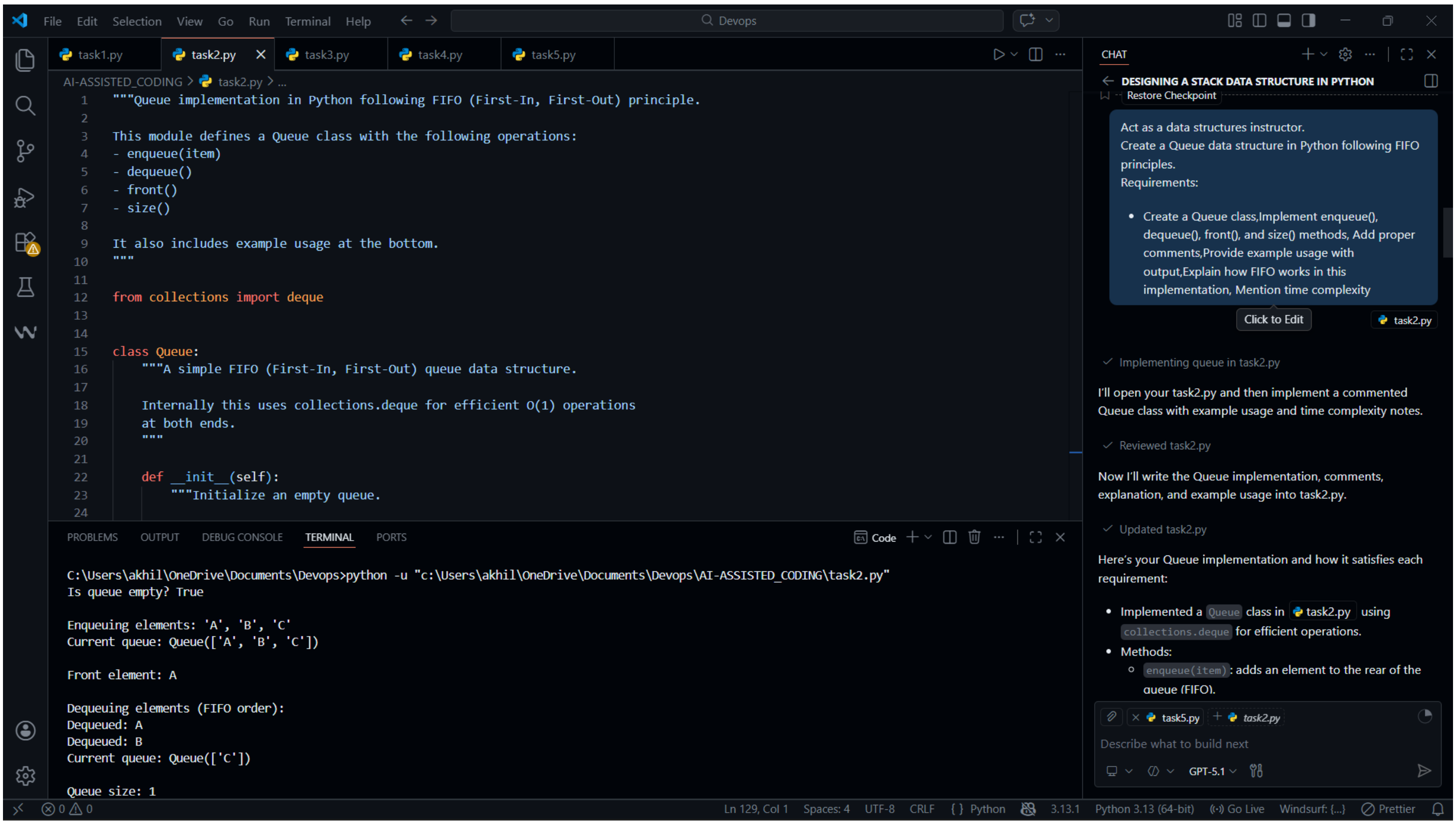
- push(item): adds an element to the top.
- pop(): removes and returns the top element, raises IndexError if empty.
- peek(): returns (without removing) the top element, raises IndexError if empty.
  is_empty(): returns True if there are no elements.

Task Description -2 – (Queue Design)

• Task:Use AI assistance to create a Queue data structure following FIFO principles

Expected Output:

• A complete Queue implementation including enqueue, dequeue, front element access, and size calculation



- Implemented a Queue class using collections.deque for efficient operations.
- Methods:

  enqueue(item): adds an element to the rear of the queue (FIFO).

  dequeue(): removes and returns the front element; raises IndexError if empty.

  front(): returns (without removing) the front element; raises IndexError if empty.

  size(): returns the current number of elements.

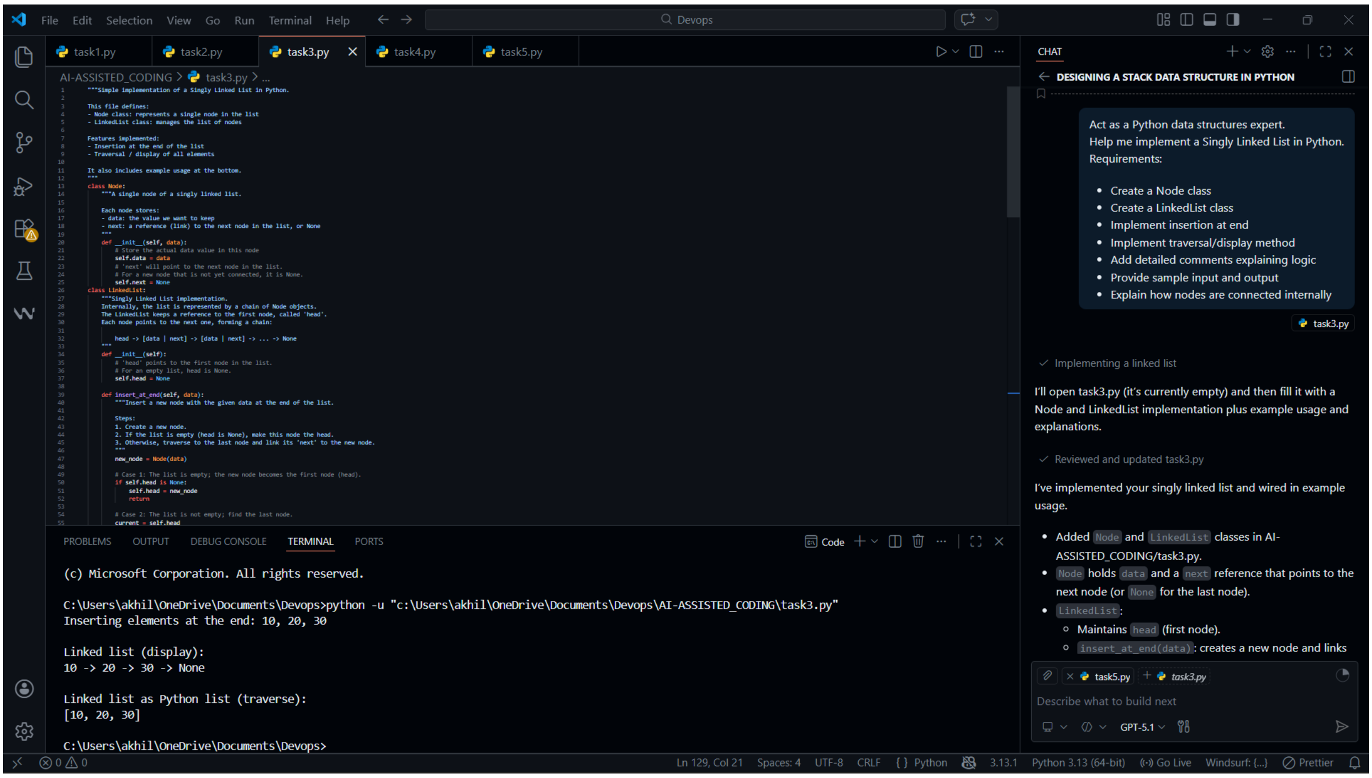
  is_empty(): helper to check if the queue has no elements.

Task Description -3 – (Singly Linked List Construction)

• Task: Utilize AI to build a singly linked list supporting insertion and traversal.

Expected Output:

• Correctly functioning linked list with node creation, insertion logic, and display functionality.



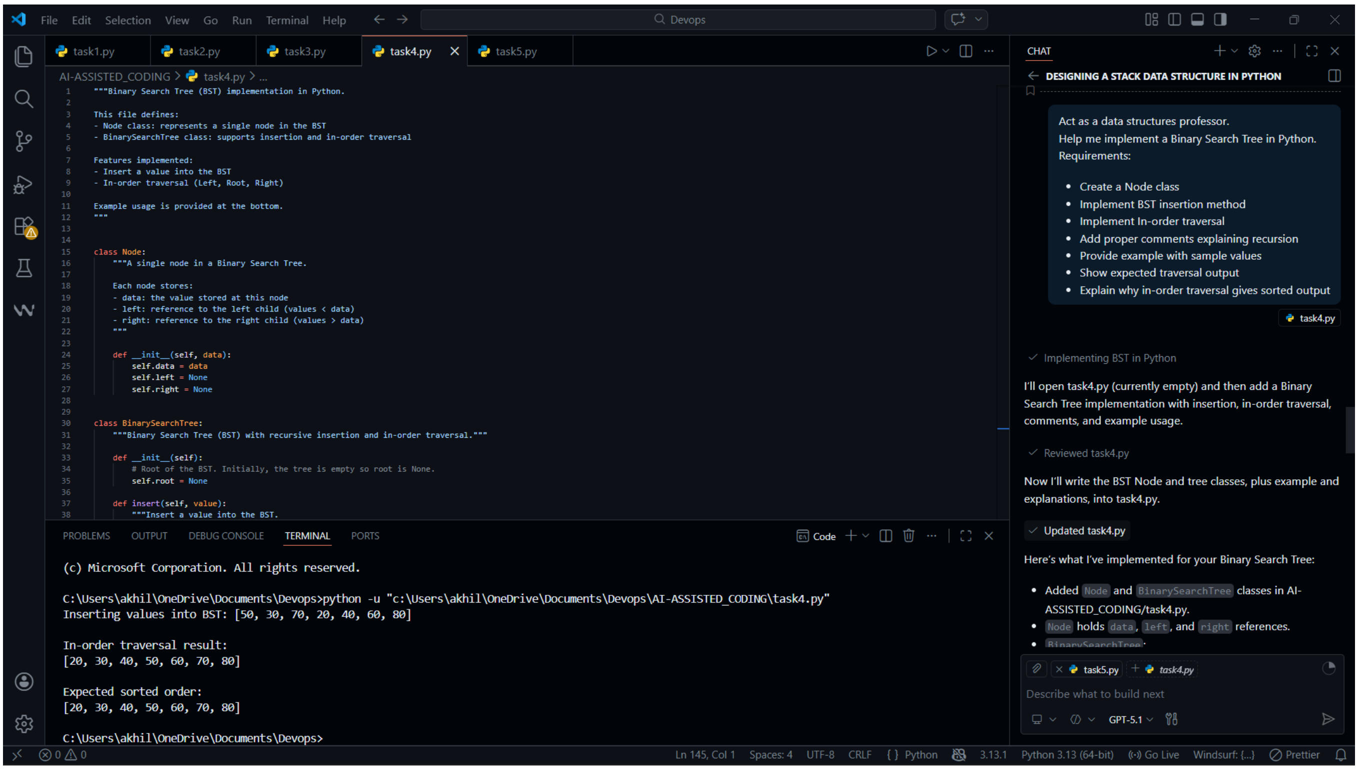
- Added Node and LinkedList classes.
- Node holds data and a next reference that points to the next node (or None for the last node).
- LinkedList:
  - Maintains head (first node).
  - insert_at_end(data): creates a new node and links it at the end by walking from head to the last node and updating its next.
  - traverse(): walks from head via next, collects data values into a Python list, and returns it.
  - display(): prints the list in the form 10 -> 20 -> 30 -> None.

Task Description -4 – (Binary Search Tree Operations)

• Task: Implement a Binary Search Tree with AI support focusing on insertion and traversal.

Expected Output:

• BST program with correct node insertion and in-order traversal output.



Added Node and BinarySearchTree classes in AI-ASSISTED_CODING/task4.py.

Node holds data, left, and right references.

BinarySearchTree:

insert(value): public insert method using _insert_recursive(node, value):

Base case: if node is None, create and return a new Node.

Recursive case: go left if value < node.data, right if value > node.data, then return node to maintain links.

inorder_traversal(): returns a list of values; uses _inorder_recursive(node, result):

Base case: node is None → return.

Recursive case: traverse left, visit node (append data), traverse right.

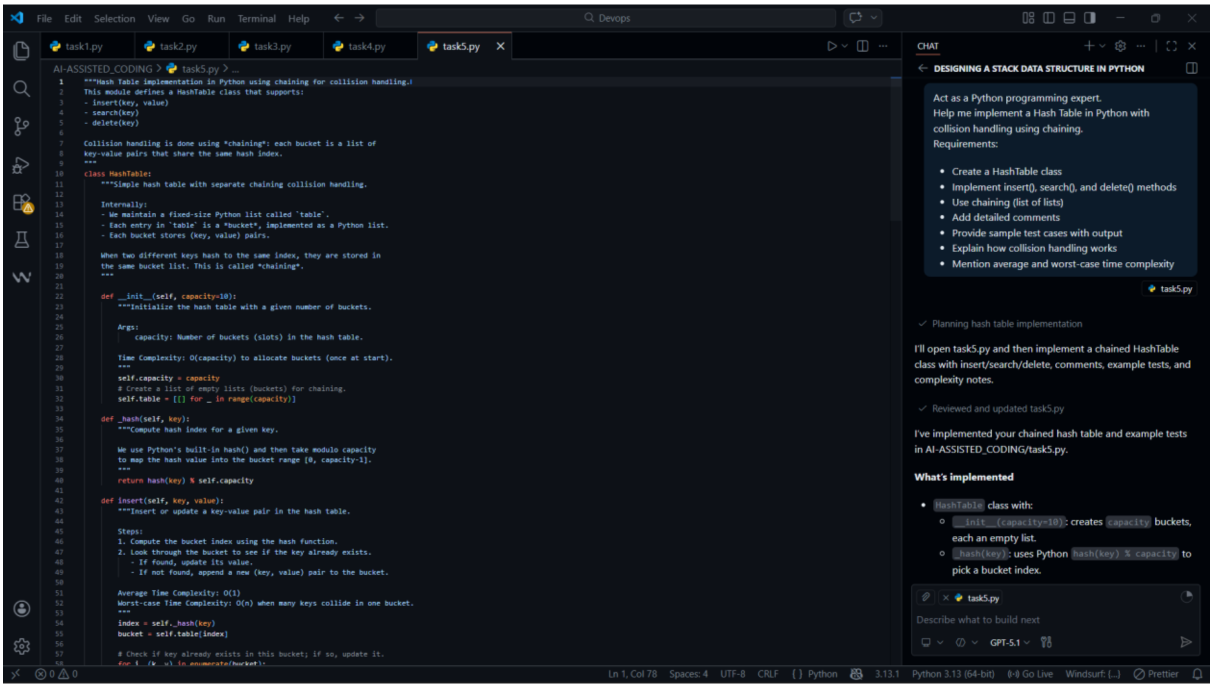
Comments in both recursive helpers explain base/recursive cases clearly.

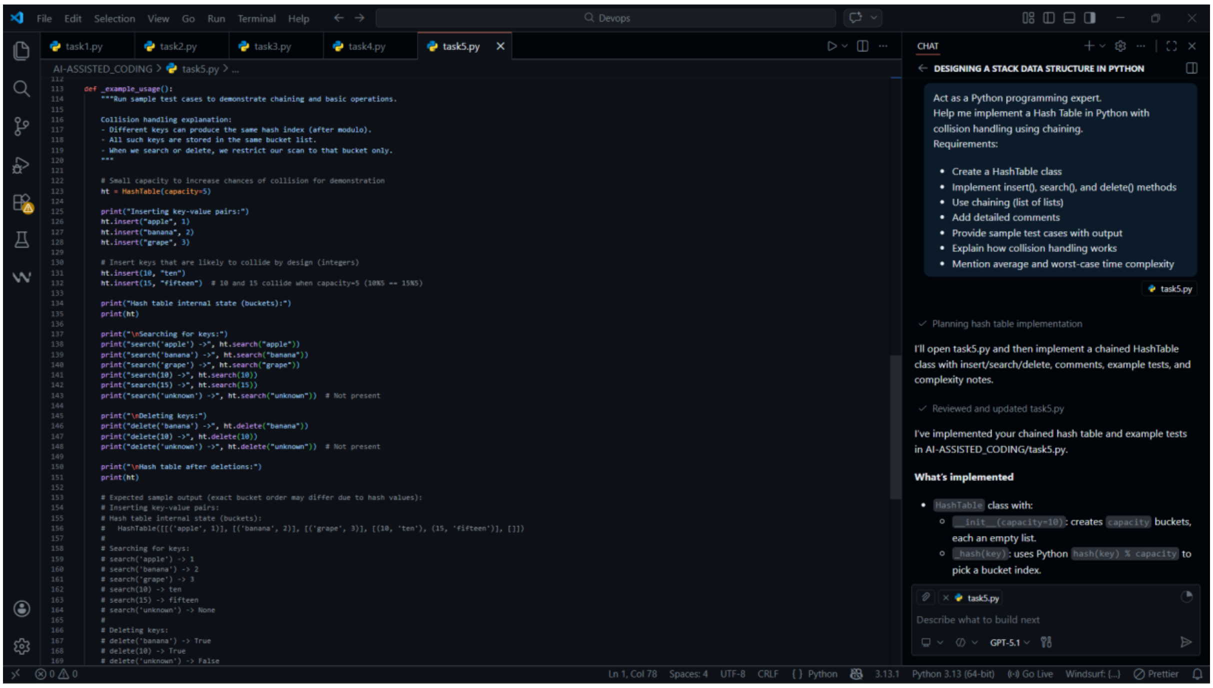
Task Description -5 – (Hash Table Implementation)

• Task: Create a hash table using AI with collision handling

Expected Output:

• Hash table supporting insert, search, and delete using chaining or open





```
Inserting values into BST: [50, 30, 70, 20, 40, 60, 80]

In-order traversal result:
[20, 30, 40, 50, 60, 70, 80]

Expected sorted order:
[20, 30, 40, 50, 60, 70, 80]

In-order traversal result:
[20, 30, 40, 50, 60, 70, 80]

Expected sorted order:
[20, 30, 40, 50, 60, 70, 80]
Expected sorted order:
[20, 30, 40, 50, 60, 70, 80]

C:\Users\akhil\OneDrive\Documents\Devops>python -u "c:\Users\akhil\OneDrive\Documents\Devops\AI-ASSISTED_CODING\task5.py"
Inserting key-value pairs:
Hash table internal state (buckets):
HashTable([[('apple', 1), (10, 'ten'), (15, 'fifteen')], [], [], [('grape', 3)], [('banana', 2)]])

Searching for keys:
search('apple') -> 1
search('banana') -> 2
search('grape') -> 3
search(10) -> ten
search(15) -> fifteen
search('unknown') -> None

Deleting keys:
delete('banana') -> True
delete(10) -> True
delete('unknown') -> False
```

- HashTable class with:
    - __init__(capacity=10): creates capacity buckets, each an empty list.
    - _hash(key): uses Python hash(key) % capacity to pick a bucket index.
    - insert(key, value): updates existing key or appends (key, value) into the bucket.
    - search(key): scans the bucket for key, returns the value or None.
    - delete(key): removes (key, value) from the bucket, returns True/False.
- Chaining:
    - self.table is a list of lists (buckets).
    - Each bucket stores multiple (key, value) pairs that share the same index → this is collision handling by chaining.